EP414 Instrumentation Design
Laboratory Manual

W. Eric Norum

October 23, 2002

1 Introduction

The microprocessor used in these labs is the Motorola MC68360 Quad Integrated Communications Controller. The MC68360 is a 32-bit microcontroller with an internal RISC communications coprocessor and includes four serial communications controllers, two serial management controllers, a serial peripheral interface, four baud rate generators, four timers and several general-purpose I/O lines. The serial communications controllers support a number of protocols including HDLC/SDLC, Appletalk and Ethernet. The memory controller on the MC68360 provides 'glueless' (i.e. no extra interface chips required) interface to dynamic RAM single-inline memory modules, static RAM, flash memory and EPROM devices. The processor core in the MC68360 is the Motorola CPU32+ which is a member of the 68000 family of processors. An 2 kbyte on-chip dual-port memory is used to transfer information between the CPU32+ core and the RISC communication controller.

The MC68360 is a very powerful microprocessor Unfortunately, this flexibility comes at the cost of complexity and it will take some time to become comfortable with all the features. The first two labs should be undertaken in 'cookbook' fashion. Just follow the steps and leave understanding the details until the lab has been completed.

All programming of the MC68360 will be done in C. A cross-compiler will translate your program to Motorola 68000 assembly language, assemble it and link it with the RTEMS real-time executive. You then use the GNU debugger (gdb) with the background debug mode (BDM) interface to load your program into the MC68360 and control its execution.

The single-board computer you will use in these labs was developed at the Saskatchewan Accelerator Laboratory and includes a MC68360 processor, a 256 kbyte bootstrap PROM, a 4 Mbyte SIMM, an RS-232 serial interface, a 10Base2 Ethernet interface and a 16-bit external I/O bus. All MC68360 I/O signals are brought to headers for connection to your circuits.

NOTE

1. ALWAYS support the circuit board near the connector when plugging cables into the board or unplugging cables from the board. There are several surface mount components installed on the bottom of the board and flexing the circuit board can crack components or connections.
2. NEVER wire wrap directly to the header pins on the single-board computer. Wrapping and unwrapping wires damages the edges of the pins.

1.1 The GCC cross-compiler

The tool which converts your program from C statements to machine code is known as a cross-compiler. The term cross-compiler is used because the compiler is running one one CPU architecture (in this case Intel Pentium) and generating machine code for another architecture (the MC68360).

The steps in the conversion process are shown in figure 1.

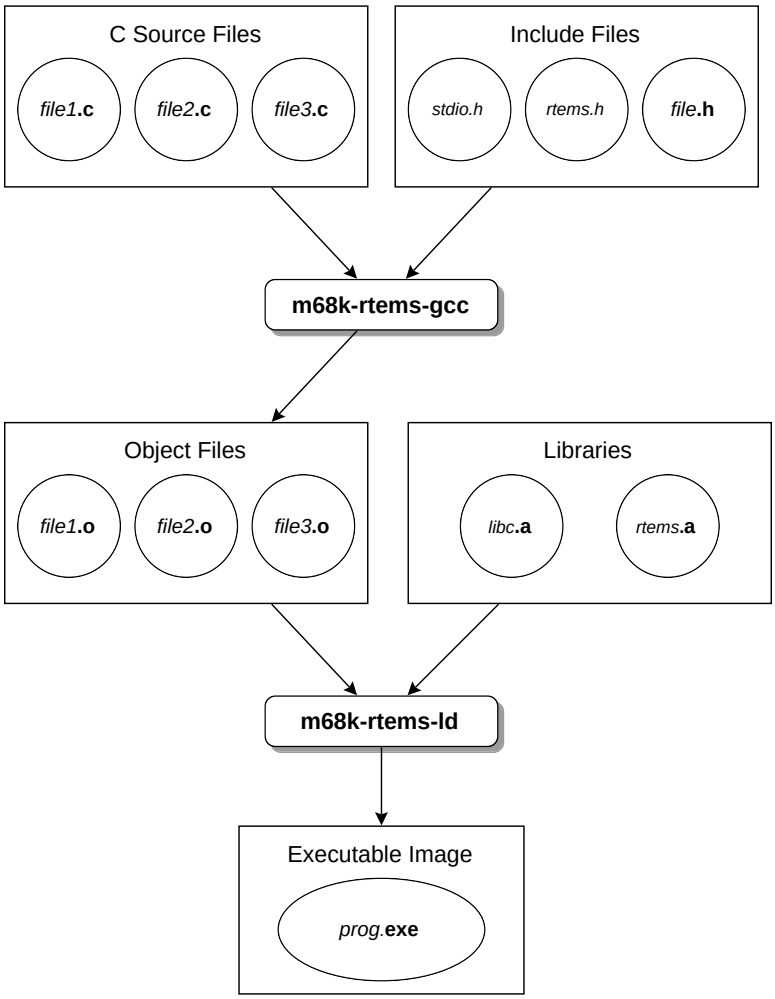


Figure 1: Cross-compilation processes

The cross-compiler (`m68k-rtems-gcc`) reads the files containing your C program (`file`*x*`.c`) and any files which it includes (`#include`) and converts the C statements to MC68360 machine code. The conversion process actually involves three separate tasks:

- The preprocessor gathers all the include files and expands all macro definitions.
- The compiler converts the resulting C statements to MC68360 assembler language.
- The assembler converts the MC68360 assembler language instructions to MC68360 machine code.

The resulting machine code is contained in 'object files'. There is one object file (`file`*x*.**o**) for each C source file (`file`*x*.**c**).

The cross-linker (`m68k-rtems-ld`) merges the RTEMS executive, the object files produced from your C source files and library routines called by your C code into a single executable image. This image is a standalone program which can be downloaded to, and executed by, the MC68360.

1.2 GNU Make

There are a very large number of commands and options that must be issued to cross-compile and link your program into an MC68360 executable image. To simplify this task the **make** utility will be used to automate this process. The **make** utility determines which pieces of your program need to be recompiled and issues the commands to recompile them. A file called the *makefile* describes the relationships among the files in your program and states the commands for updating each file. Once a suitable makefile exists, each time you change some source files, the command:

```
make
```

will perform all the steps required to rebuild the executable image.

Command-line arguments to `make` can be used to control which files should be recompiled, or how. For example, the command:

```
make debug
```

will build a version of your program suitable for full operation under the GNU cross-debugger.

1.3 RTEMS

The application programs which you will write in these laboratory sessions will use RTEMS (Real-Time Executive for Multiprocessor Systems) to provide real-time, multitasking support. The RTEMS documentation is available in html format. The C User's Guide will be of most interest to you, in particular the overview, the key concepts, and the sections on tasks, events, and interrupts. A pdf version of the RTEMS C user's guidede is also available.

2 Lab 1 - Introduction to GCC and GDB

This lab is an introduction to some of the software development tools that will be used in the EP414 laboratory sessions. It will also give you practice in writing and testing C programs.

2.1 The Lab

You are to write and compile three simple C programs and run them on the workstations in the EP414 lab.

1. Write, compile and execute the classical first C program. The program should print `Hello world.` (followed by a newline).

2. Write a C program consisting of a subroutine and a main program. The subroutine should take 3 double-precision floating-point arguments. The subroutine should compute and print the roots of the quadratic expression with these three numbers as the coefficients. The roots should be printed to 5 significant figures.

   The main routine should prompt for three numbers then call the subroutine with the values entered by the user.

   Compile your program and use GDB to control its execution. Try placing a breakpoint on your subroutine entry point and single step through the subroutine. Use the GDB `print` command to print the values of the arguments to the subroutine and variables within the subroutine.

3. Write a C program consisting of a signal-handling subroutine and a main routine. The main routine should call `signal()` to arrange that the signal-handling subroutine is called whenever an 'interrupt' (control-C) character is typed. The main routine should then busy loop until the value of an external integer variable (`interruptCount`) reaches 5. The main routine should then print `ALL DONE!` (followed by a newline) and exit.

   The signal handling subroutine should increment the `interruptCount` variable and arrange to continue catching 'interrupt' signals.

   Compile and run your program without optimization (no compiler **-O2** option) and with optimization. Verify that it works properly in both cases.

All programs must compile with no warning messages even when all warning messages are enabled (compiler **-Wall** option). All programs must run properly when compiled with the optimizer enabled (compiler **-O2** option).

Remember to use the compiler **-g** option to ensure you can use the debugger to test your programs

2.2 Questions

1. How big are the object files produced from your source file? (Hint: Use the compiler **-c** option and the `size` command).

2. How big is the final executable image? If your program were to be burned into PROM how much space in the PROM would it use? How much space in RAM would it use when it ran?

3. How would you modify the root-finding program to allow the numbers to be entered as command-line arguments?

3 Lab 2 - Introduction to the Motorola MC68360

This lab is an introduction to the Motorola MC68360 kits and the software development tools that will be used in the EP414 laboratory sessions. Because the development tools and hardware connections are relatively complex the instructions are quite detailed. Don't panic if you don't fully understand why you're doing some of the steps. You should, however, have some idea of what you are accomplishing with each of the steps.

3.1 Prerequisites

1. You should be able create, compile and execute a simple C program.
2. You should be able to use the GNU debugger. to control the execution of another program:
   - Set breakpoints
   - Start execution
   - Single step
   - Print variables
3. You should have read the *Overview* and *Introduction* chapters of the GNU make manual and have some idea of `make`'s place in the software development process.
4. You should know how to use `kermit` to communicate with devices attached to the PC serial ports.

3.2 Step 1. Check the hardware connections

- Check that the BDM adapter is connected to the parallel printer port on the computer and to the BDM connector on the MC68360 development board.
- Check that the serial port on the MC68360 development board is connected to serial port B (`COM2:`, `/dev/ttyS1`) on the computer.

3.3 Step 2. Verify that BDM works

- Bring up an xterm window and start kermit. Check that the kermit settings are:
  - `baud rate = 9600`
  - `parity = none`
  - `port = /dev/ttyS1`

  You might want to edit your `~/.kermrc` file to reflect the above settings.
- Once the settings are correct, use the kermit `connect` command (which may be abbreviated to just `c`) to 'connect' kermit to the MC68360 development board.
- Apply power to the MC68360 development board.
- Bring up another xterm window and issue the command

  ```
  m68k-rtems-gdb /opt/rtems/m68k-rtems/gen68360/samples/hello.exe
  ```

  This should produce the GDB startup message and a `(gdb)` prompt:

  ```
  GNU gdb 4.18
  Copyright 1998 Free Software Foundation, Inc.
  GDB is free software, covered by the GNU General Public License, and you are
  welcome to change it and/or distribute copies of it under certain conditions.
  Type "show copying" to see the conditions.
  There is absolutely no warranty for GDB.  Type "show warranty" for details.
  This GDB was configured as "--host=i586-pc-linux-gnu --target=m68k-bdm-aout"...
  ```

```
Breakpoint 1 at 0x400
Breakpoint 2 at 0x5dee: file error.c, line 207.
Breakpoint 3 at 0x130a: file interr.c, line 50.
Breakpoint 4 at 0x1fd8: file fatal.c, line 35.
Breakpoint 5 at 0x4a4
(gdb)
```

- Type `run` followed by a carriage return. This should produce the following question:

  ```
  Starting program: /opt/rtems/m68k-rtems/gen68360/samples/hello.exe
  Do you want to download '/opt/rtems/m68k-rtems/gen68360/samples/hello.exe'?(y or n)
  ```

- Type `y` followed by a carriage return.
- The `hello.exe` program will be downloaded and run on the MC68360 development board. In the kermit window you should see:

  ```
  *** HELLO WORLD TEST ***
  Hello World
  *** END OF HELLO WORLD TEST ***
  ```

- When the `hello.exe` program finishes you should see

  ```
  Breakpoint 5, 0x4a4 in _mainDone ()
  ```

  in the gdb window and gdb will terminate.

3.4 Step 3. Compile and run an RTEMS application

- Create a new directory (`mkdir`) and `cd` to it.
- Copy the 'boilerplate' files Makefile lab2.h and init.c to your newly-created directory.
- Write your application code. You source file must be called `subr.c` and must provide the function:

  ```
  void printMessage (const char *message)
  ```

  This function should print its argument so that the output appears like:

  `The message is '`*the message argument*`'.`

  where *the message argument* is the string pointed to by the argument to the function.

  Remember to `#include` the appropriate header files for any functions called by your code.

- Compile a 'debugging' version of your application by:

  ```
  make debug
  ```

  Make sure your application compiles with no warning messages.

  You will see from the lengths of the commands why it is not practical to issue the commands directly!

  The object files and executable image generated from your source code are placed in the `o-debug` directory.

- Use GDB to download and run your application:

  ```
  m68k-rtems-gdb o-debug/lab2.exe
  ```

3.5 Questions

1. What are the sizes of code (`text`), initialized data (`data`) and uninitialized data (`bss`) sections of the object file produced from your C source file and of the resulting executable image? Hint: Use the m68k-rtems-size program.
2. Give an estimate (in bytes per second) of the speed at which the executable image is transferred to the MC68360 development board?
3. What line in the `Makefile` would you change if you wanted to change the name of your source file to `lab2.c`?

4 Lab 3 - Simple I/O and Multitasking

The MC68360 uses memory-mapped I/O for all device registers. Through some trickery in the linker script used to build RTEMS applications the device registers appear as elements of the C structure `m360`. Use

```
#include <bsp.h>
#include <m68360.h>
```

to make the structure and associated macros available to your program. As far as possible, the names of the structure elements match (except for being in lower case) the register names in the MC68360 data book. For example, to read the Port A Data Register (`PADAT`) into the variable `i`:

```
i = m360.padat;
```

The structure is declared in

```
/opt/rtems/m68k-rtems/gen68360/lib/include/m68360.h
```

If the size of the MC68360 data book didn't convince you that this is a complicated device, the number of device registers should! Notice that `m360` is declared as a `volatile` structure. The `volatile` modifier tells the compiler that the values in the structure may change in ways not apparent from the C statements. The compiler uses this information to inhibit some 'optimizations' that could lead to undesired operation of the program.

4.1 Prerequisites

1. Read Section 7.14 (Parallel I/O Ports – pages 7-356 to 7-369) of the Motorola MC68360 User's Manual.
2. Read the introductory chapters of the RTEMS C User's Guide:
   - Preface
   - Overview
   - Key Concepts
   - Task Manager
   - Event Manager
3. Read the description of the input/output signals and connectors in the SAL embedded controller manual.

4.2 The Lab

1. Create a new directory and `cd` to it.
2. Copy the 'boilerplate' files Makefile and init.c to the new directory.
3. Edit the `Makefile` so the application:
   - is built from three source files, `init.c`, `taska.c` and `taskb.c`.
   - produces the executable image `lab3.exe`.
4. Modify `init.c` so that the initialization task, instead of calling `printMessage`, starts `taskA` and `taskB` then suspends itself. The tasks should have the following attributes and modes:
   - Local
   - Non-floating point
   - Preemption enabled
   - Timeslicing disabled
   - ASR processing disabled
   - Interrupt level 0.
   - Task A should run at priority 100.
   - Task B should run at priority 50.

   RTEMS provides the `rtems_build_name` macro for creating names. The macro takes four character arguments and uses them to build an RTEMS name.

   Almost all RTEMS routines return a status code (type `rtems_status_code`). You should always check this code and ensure that its value is `RTEMS_SUCCESSFUL`. If it is not, you should print an error message. The `rtems_status_text` function returns a string describing the status code.

   The following code fragment shows how RTEMS names are built, tasks are created, and errors are detected and reported.

```
#include <rtems/error.h>
   .
   .
   rtems_status_code sc;
   rtems_id tid;
   .
   .
   sc = rtems_task_create (rtems_build_name ('T', 'S', 'K', 'A'),
                50,
                8*1024,
                RTEMS_PREEMPT|RTEMS_NO_TIMESLICE|RTEMS_NO_ASR|
                                    RTEMS_INTERRUPT_LEVEL(0),
                RTEMS_NO_FLOATING_POINT|RTEMS_LOCAL,
                &tid);
   if (sc != RTEMS_SUCCESSFUL) {
        printf ("Can't create task: %s\n", rtems_status_text (sc));
        rtems_task_suspend (RTEMS_SELF);
   }
   .
   .
```

5. In the `taska.c` source file, write an RTEMS task with entry point `taskA`. This task should:

   (a) Get the task ID of task B. There are several ways to do this.
   - Have the initialization task pass the task B ID as the argument to task A.
   - Have task A look up task B by name (`rtems_task_ident`).
   - Have the initialization task put the task B ID in a global variable.
   - Have the initialization task put the task B ID in a task A notepad (`rtems_set_note`).

   (b) Set up the Port A control registers to configure an unused Port A pin as an actively-driven output. Always use the C 'bit-set' and 'bit-reset' operators when manipulating the device registers. This ensures that your code does not modify other bits in the device registers.

   (c) Enter an infinite loop of the following operations.

   (d) Wait for an RTEMS clock tick (`rtems_task_wake_after`).

   (e) Drive the Port A output pin HIGH.

   (f) Send an event to task B.

   (g) Drive the Port A output pin LOW.

6. In the `taskb.c` source file, write an RTEMS task with entry point `taskB`. This task should:

   (a) Set up the Port B control registers to configure an unused Port B pin as an actively-driven output.

   (b) Enter an infinite loop of the following operations.

   (c) Wait for an event (the same event number as sent by task A).

   (d) Toggle the state of the Port B output pin.
   Hint: Remember the C `^=` operator.

7. Compile and execute your program. Use the oscilloscope to view the Port A and Port B pins.

   If you are having difficulty with your program keep in mind that GDB is a very useful and powerful debugging tool. The GDB startup script sets the internal GDB variable `$m360` to be a pointer to the MC68360 dual-port RAM and I/O registers. For example, the GDB command `print/x $m360->papar` will print the Port A Pin-Assignment Register (PAPAR) in hexadecimal format.

8. Modify your program so that task B has a lower priority than task A. Use the oscilloscope to view the Port A and Port B pins.

9. Modify your program so that task B has the same priority as task A. Use the oscilloscope to view the Port A and Port B pins.

4.3 Questions

1. What is the 'tick' rate of the RTEMS clock?

2. How long does it take for the event from task A to awaken task B?

3. How long does it take for task B to block and execution of task A to resume?

4. Are the above times affected by compiling your application without the `debug` option?

5. Explain the oscilloscope displays for the different task B priorities. What do they tell you about the RTEMS task scheduling algorithm?

5 Lab 4 - Serial Peripheral Interface Analog to Digital Conversion

In this lab you will begin to use some of the multiprocessing features of the MC68360 to build a complete 8-input, 12-bit analog data acquisition system.

5.1 Direct Memory Access Input/Output

The MC68360 includes two processors. The first is the CPU32+ core processor which takes care of general purpose computation and the second is the RISC communications processor which operates the various I/O devices on the chip. In this lab you will acquire data from an analog to digital converter chip using the RISC communications processor module (CPM) to control the data transfer.

The RISC communications processor and the CPU32+ processor communicate with each other through an on-chip block of memory called the Dual-Port RAM (DPRAM). Within the DPRAM are 'Parameter RAM' areas for each of the I/O devices and space for 'Buffer Descriptors' which specify where data is to be transferred to and from main memory and the I/O devices. Each I/O device also has some control and status registers which are read and written by the CPU32+ to control the operation of the device. This I/O system is very powerful in that the CPU32+ processor is not burdened with the task of copying data between system memory and I/O device registers. Instead the CPU32+ processor merely indicates the location and size of data to be transferred to or from the I/O device registers. The actual transfer is handled by the RISC communications processor using a mechanism known as Direct Memory Access (DMA). This name arises from the fact that I/O devices transfer data directly to and from system memory without processor intervention. Entire blocks of data can be transferred with a single operation. For example, the set of commands to acquire data from all the MAX186 channels can be sent as one operation.

The CPU32+ indicates the location and size of the data transfer locations in main memory by placing that information in 'buffer descriptors' stored in the DPRAM. The CPM uses the information in the buffer descriptors to transfer data between the specified locations in main memory and the I/O devices. Each buffer descriptor is eight bytes long and has the following structure:

```
typedef struct m360BufferDescriptor_ {
        unsigned short          status;
        unsigned short          length;
        volatile void           *buffer;
} m360BufferDescriptor_t;
```

- The `status` field contains bits which define the current status of the buffer descriptor. Among these are a bit which indicates which processor (CPU32+ or CPM) has control of the buffer descriptor, and a bit which indicates whether or not the CPM should set a bit in the device event register when the data associated with the buffer descriptor have been transferred to or from the I/O device.
- The `length` field is set by the CPU32+ to specify the number of bytes to be transferred from main memory for a write operation or by the CPM to indicated the number of bytes transferred to main memory by a read operation. Since this field is only 16 bits long the maximum transfer for a single buffer descriptor is 65535 bytes.
- The `buffer` field indicates the location in main memory at which the transfer should begin.

Since buffer descriptors are accessed by both the CPU32+ and the CPM, they must reside in the on-chip dual-port memory. The `M360AllocateBufferDescriptors` function sets aside space in the dual-port

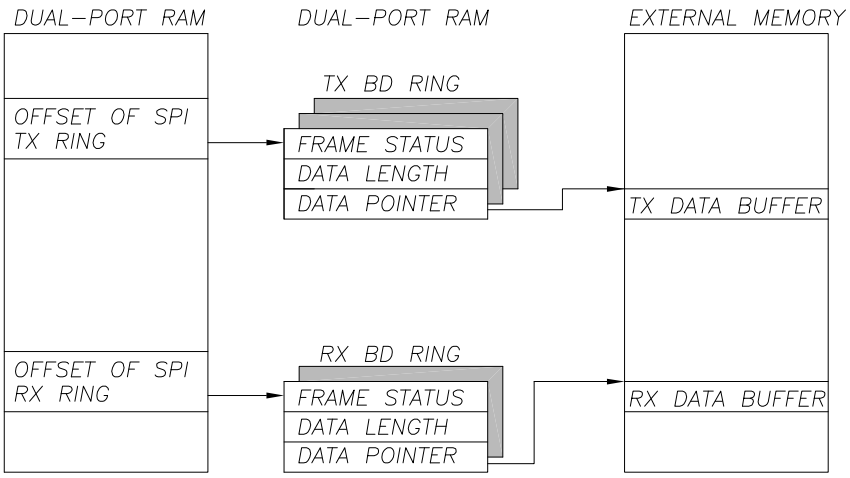


Figure 2: SPI Buffer Descriptor Structure

memory for the specified number of buffer descriptors and returns a pointer to the space. The CPU32+ processor accesses the allocated buffer descriptors using this pointer.

The CPM must also know where the buffer descriptors for a given I/O device are located. Each I/O device has a predefined section of dual-port RAM which is used to hold various I/O parameters associated with that device. Among these parameters are 16-bit values indicating the offset into the dual-port RAM where the transmit and receive buffer descriptors for that device have been allocated. The CPU32+ sets up the parameter values in the dual-port RAM then writes a command (`M360_CR_OP_INIT_RX_TX`) to the CPM command register telling the CPM to use the specified parameters.

Since there may be more than one buffer descriptor allocated for a given input or output device the CPM must know when it has reached the device's final descriptor. This is done by setting a bit in the buffer descriptor status word which indicates that that descriptor is the last one allocated for that device. When the CPM finishes the I/O associated with a buffer descriptor it checks this bit and either moves to the next higher buffer descriptor in memory or it 'wraps around' to reuse the first buffer descriptor associated with the device.

Figure 2 shows the relationship between the device parameters in dual-port memory, the buffer descriptors in dual-port memory, and the I/O buffers in main memory for the SPI device.

Each I/O device has a set of control registers in addition to its parameter storage area in the dual-port RAM. The SPI, for example, has:

SPMODE to set the operating modes of the SPI interface.

SPCOM to start an SPI transmit/receive operation.

SPCOM to start an SPI transmit/receive operation.

SPIE to report events recognized by the SPI and to generate interrupts.

SPIM to control which events in the SPIE cause an interrupt request to the CPU32+.

5.2 The MAX186 analog to digital converter

The MAX186 analog to digital converter chip is a complete 8-channel, 12-bit, analog input system on a single chip. The MAX186 interfaces to the the Serial Peripheral Interface (SPI) on the MC68360 using only three lines:

SPICLK SPI Clock output from the MC68360. The MAX186 reads the value at the DIN pin on the rising edge of this signal and shifts the next bit to the DOUT pin on the falling edge of this signal.

SPIMOSI SPI Master-Out Slave-In. This line carries control information from the MC68360 to the MAX186 DIN pin.

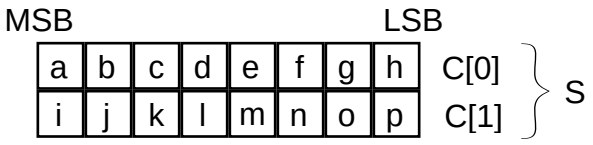
SPIMISO SPI Master-In Slave-Out. This line carries the acquired data from the MAX186 DOUT pin to the MC68360.

The SPI, like all the other I/O devices in the M68360, shares its I/O pins with the parallel ports. The corresponding bits in the PBPAR and PBDIR registers must be programmed appropriately to assign the I/O pins to the SPI functions. See Table 7-20 in the Motorola MC68360 User's Manual.

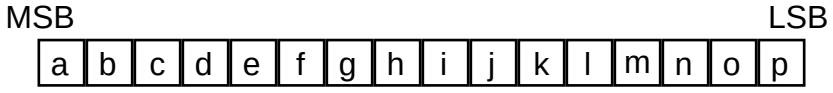
The MC68360 User's Manual is not very clear in its discussion of the order in which bits are transmitted and received on the serial peripheral interface. Here is the way bits are actually transferred. The following C union allows memory to be accessed as a 16-bit value (`sc.s`) or as two 8-bit values (`sc.c[0]` and `sc.c[1]`).

```
union scunion {
        unsigned char  c[2];
        unsigned short s;
} sc;
```

Given such a union containing the following pair of 8-bit values,

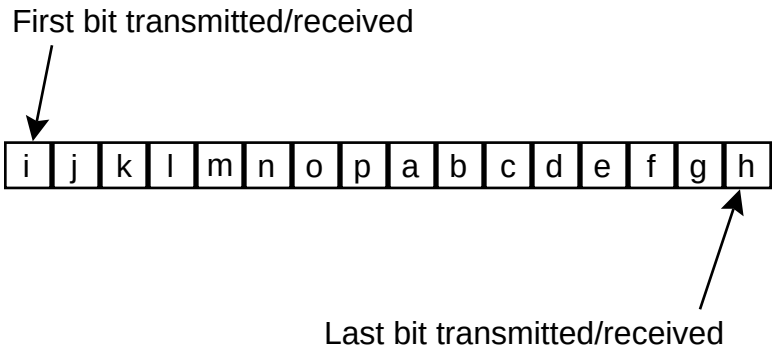


the value of `sc.s` would be:



This byte order shows that the MC68360 is a 'big-endian' machine. On big-endian machines the lower memory address (`sc.c[0]`) holds the most-significant byte of the 16-bit value.

The SPI, programmed for 16-bit transfers, would transmit `sc.s` as:



This is certainly not very intuitive, but that's the way the hardware works. Keep this in mind when setting up the commands to be transferred to the MAX186 and when manipulating the values received from the MAX186.

5.3 Joystick Connections

The analog joysticks used in this lab consist of two potentiometers whose wipers are moved by the X and Y motion of the stick. The joystick connections are shown in figure 3.

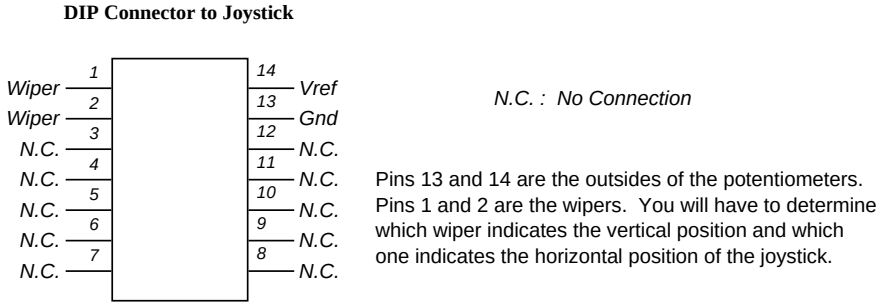


Figure 3: Analog joystick connections

5.4 Prerequisites

1. Read the data sheet for the MAX186 analog to digital converter.
2. Read Section 7.12 (Serial Peripheral Interface (SPI)) of the Motorola MC68360 User's Manual.

5.5 The Lab

1. Construct the circuit shown in figure 4.
2. Write a C function which will set up the MC68360 SPI to communicate with the MAX186. There are quite a few things that need to be configured. Here is the list of things your function must do:
   - Disable the SPI (clear the `M360_SPMODE_EN` bit in `m360.spmode`).

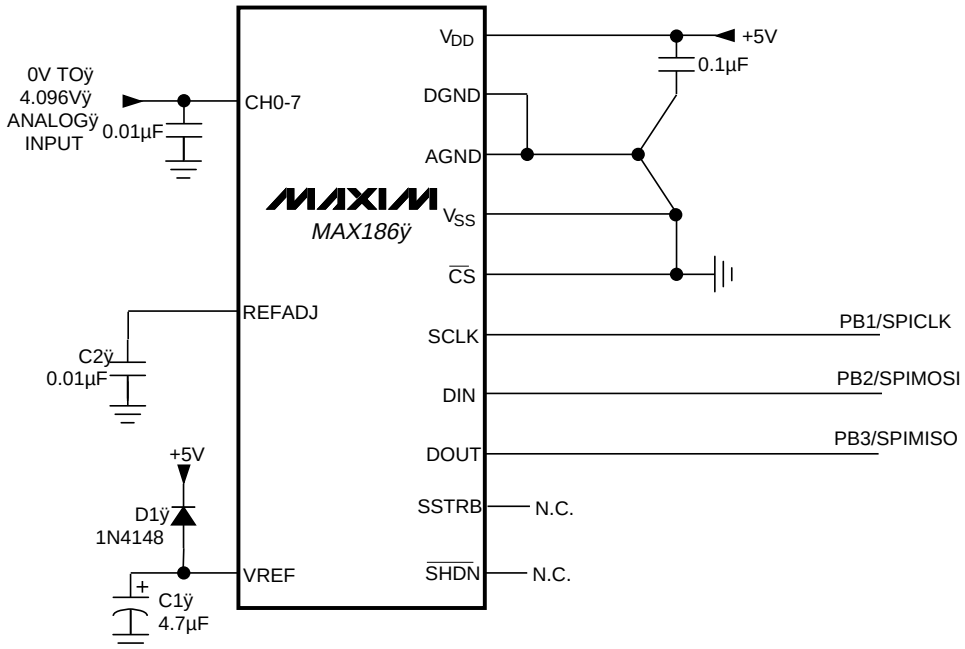


Figure 4: MAX186 A/D Converter Wiring

- Configure the appropriate Port B pins for SPI I/O. Use C bit-set and bit-clear operators on the configuration registers.
- Allocate a transmit and a receive buffer descriptor. The buffer descriptors should be declared as:

  ```
  volatile m360BufferDescriptor_t *spiRxBd, *spiTxBd;
  ```

  The function

  ```
  void *M360AllocateBufferDescriptors (int count);
  ```

  should be used to allocate the buffer descriptors.
- Set up the SPI parameter RAM. The RBASE and TBASE entries, which specify the offset into the dual-port memory to the first (and in this case) only buffer descriptors for the SPI receiver and transmitter, are computed from the allocated buffer descriptors as:

  ```
  m360.spip.rbase = (char *)spiRxBd - (char *)&m360;
  m360.spip.tbase = (char *)spiTxBd - (char *)&m360;
  ```

  The `(char *)` casts force the compiler to compute the difference as the number of bytes between the locations.

  The receive and transmit function codes should be set as:

  ```
  m360.spip.rfcr = M360_RFCR_MOT | M360_RFCR_DMA_SPACE;
  m360.spip.tfcr = M360_TFCR_MOT | M360_TFCR_DMA_SPACE;
  ```

  The maximum receive buffer length parameter should be set to 18 (2 bytes per MAX186 channel plus 1 byte for the initial command all rounded up to an even number of bytes).
- Tell the RISC processor to start using the values in the SPI parameter RAM by:

  ```
  M360ExecuteRISC (M360_CR_OP_INIT_RX_TX | M360_CR_CHAN_SPI);
  ```

This function sends the command to the RISC processor and waits until the command has been accepted.

- Clear all the bits in the SPI Event register. Read the manual (Section 7.12.5.6) carefully to see how bits in this register are cleared. Hint: You don't clear the register by writing a 0 to it!
- Configure the SPI. The SPI mode should be:
  - Inactive state of SPICLK is low.
  - SPICLK should begin toggling in the middle of the data transfer.
  - SPI data rate should be around 480 kbits/second (k=1000).
  - SPI data should be sent and received MSB first.
  - The MC68360 should be the SPI master.
  - The SPI should be enabled.
  - SPI data should contain 16 bits per character

3. Write a C function which will transfer data to and from the MAX186 attached to the SPI. The function should take three arguments.

   - A pointer to an array of unsigned short values which will be transferred to the MAX186.
   - A pointer to an array of unsigned short values which will be filled with the values transferred from the MAX186.
   - An integer specifying the number of unsigned short values to be transferred.

   The function should set up the transmit and receive buffer descriptors, start the SPI, then wait till the transfer completes. The function can simply poll the SPI event register to determine when the transfer is finished. Don't forget to clear the SPI event register bits after the transfer is complete!

   Hint: To test this function, connect the MC68360 SPIMOSI and SPIMISO pins together. If your function is working properly the receive buffer should contain the same values as the transmit buffer when the transfer is complete. A digital oscilloscope connected to the SPICLK and SPIMOSI/SPIMISO pins will let you see what's being transmitted and received.

4. Verify that your SPI I/O function works with transfer sizes between 2 and 30 bytes.

5. Write an RTEMS task which calls the SPI initialization function, then enters a loop reading all 8 channels of the MAX186 and printing their values around every two seconds.

6. Connect a joystick to two of the MAX186 inputs and verify that your system is operating correctly. Use the MAX186 V_{REF} output to drive the joystick potentiometers.

5.6 Questions

1. How much noise do you detect in the joystick digitized values?

2. What is the purpose of the 0.01μF capacitors on the analog input lines of the MAX186?

3. Why is it necessary to include the `volatile` keyword in the declartion of the pointers to the receive and transmit buffer descriptors (`spiRxBd` and `spiTxBd`)?

6 Lab 5 - A Simple Network Server Application

In this lab you will use the Ethernet networking features of the MC68360 to access the data acquisition system from lab 4 from any web server.

6.1 Prerequisites

1. Ensure that the data acquisition system constructed in lab 4 is operating reliably.
2. Read Section 7.15 (CPM Interrupt Controller) of the Motorola MC68360 User's Manual.
3. Read the Introduction to RTEMS TCP/IP Networking.

6.2 The Lab

1. Copy the 'boilerplate' files Makefile and init.c to your directory. You should not have to make any changes to the code in `init.c`. Your source file should be called `lab5.c` and the entry point should be:

   ```
   void
   lab5 (void)
   ```

2. Modify the data acquisition task from lab 4 so that instead of printing the converted values every two seconds, it saves them in a global array twice a second. As well, rather than polling for completion, the SPI I/O routine should use SPI interrupts and some form of task synchronization.
3. Write a `lab5` function. This function should start the data acquisition task and then enter a loop accepting connections from web clients. The data acquisition task should run at a higher priority than the web server tasks.

   (a) The server should accept connections on TCP port 80 (the standard HTTP port) from any Internet host.

   (b) When a connection is accepted, the server should print the Internet address and TCP port number of the remote host, then start an RTEMS task to communicate with the remote host.

   Hint: Pass the socket descriptor for the connection to the remote host as the argument to the RTEMS task.

   (c) The RTEMS task serving the remote host should read from the socket until it receives a newline or an end-of-file. The task should then print the client IP address and port number, the full URL which was sent to the server and the values from the A/D converter task to the socket I/O stream. The tast should then close the I/O stream and destroy itself. The line read from the client contains the HTTP request specifying the information requested by the client. Since your web server is very simple you can ignore the contents of this line.

   Hint: Use `fdopen()` to associate a `"r+"` STDIO stream with the socket descriptor then use `fgets()` to read the URL from the client, `fprintf()` to transmit the values and `fclose()` to close the stream and the socket.

   The information sent to the client must be preceded and followed by some HTML 'boilerplate'. A template of the information to be transmitted is shown below. Note that there is a blank line between the `Content-type` and `<HTML>` lines.

   Feel free to modify this template as your creative talents and knowledge of HTML allow. One interesting possiblity would be to include the numbers of the desired ADC channels as part of the URL. The server would use this information to display only the specified channels. If you're really hard-core you might try modifying the application to use 'client-pull' or 'server-push' to continuously update the values. If you're really, really, an HTML expert you could drive an applet to continously update the values.

```
HTTP/1.0 200
Content-type: text/html

<HTML>
<HEAD>
<TITLE>EP414 Data Acquisition</TITLE>
</HEAD>
<BODY>
<H1>ADC VALUES</H1>
<HR>

Connection from client: xxx.xxx.xxx.xxx:ppp
URL: URL read from client
ADC: Here's where you print the ADC values

<HR>
</BODY>
</HTML>
```

4. Compile and run the Internet server application. Test it by requesting

   `http://`*your_68360_host_name*`/`

   from your web browser. Every time you hit the `<Reload>` button you should see the converted values.

5. Verify that you can have more than one client running at a time.

6.3 Hints

The prototype for an RTEMS interrupt handler is:

```
rtems_isr interruptHandler (rtems_vector_number v);
```

The following code attaches the MC68360 SMC1 interrupt to this handler:

```
rtems_isr_entry old_handler;
.
.
sc = rtems_interrupt_catch (interruptHandler,
                            (m360.cicr & 0xE0) | 0x04,
                            &old_handler);
```

6.4 Questions

1. How would you modify the program so that it would reject connections from hosts outside the U of S domain?

2. Briefly describe how you would modify the program so that it would transmit the value of a single channel as specified by a URL of the form

   `http://`*68360_host_name*`/`*channel_number*`/`

7 Lab 6 - Variable Speed Stepper Motor Drive

In this lab your application will have the MC68360 drive a stepper motor.

7.1 Prerequisites

1. Ensure that the data acquisition system constructed in lab 4 and network server written in lab 5 are operating reliably.
2. Brush up on the operation of bipolar transistors.
3. Read Section 7.5 (Timers) of the Motorola MC68360 User's Manual.

7.2 The Lab

1. Design a driver circuit for an SM4202 stepper motor. As shown in figure 5 the SM4202 is a 100-steps/revolution, unipolar, 5 lead, 4 phase stepper motor rated at 12 volts and 150 milliamps per phase.

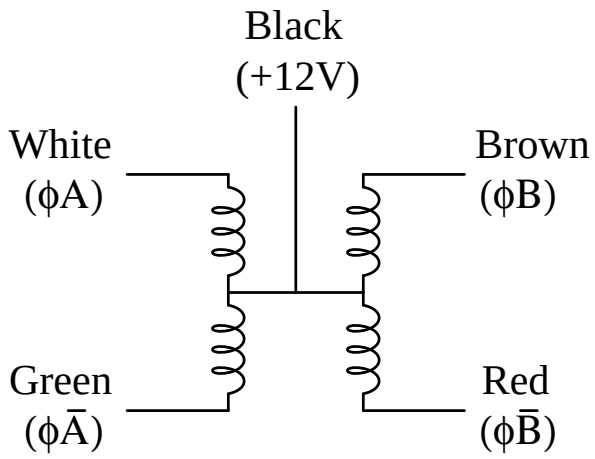


Figure 5: SM4202 Stepper Motor

For full-step operation the 4 phases should be driven as showing in table 1

Your design should use four NTE331 transistors (one per phase) to drive the motor. The NTE331 specifications are:

- Hfe is typically 40
- Vcbo 100v
- Vceo 100v
- Vebo 5v
- Ic 15amps
- Pd 90w

Table 1: Driving an SM4202 Motor

Step	A	$\overline{A}$	B	$\overline{B}$
1	ON	OFF	ON	OFF
2	OFF	ON	ON	OFF
3	OFF	ON	OFF	ON
4	ON	OFF	OFF	ON
1	ON	OFF	ON	OFF
2	OFF	ON	ON	OFF
.	.	.	.	.
.	.	.	.	.

- Ft 3MHz min

Use a 74LS05 (or 74HC05) between the MC68360 output port and the NTE331. **Do not connect the MC68360 directly to the transistors.**

Remember to include protection diodes in your circuit.

2. Have your design checked by an instructor. Remember that an incorrect design, or an improperly wired circuit can destroy the MC68360!

3. Construct the driver circuit. With some care you should be able to fit the power transistors into a wire wrap socket.

4. Modify your data acquisition software:

   - Use one of the 68360 timers to generate 10 Hz interrupts.
   - The timer interrupt handler should send an event to the data acquisition task.
   - The data acquisition task should be the highest-priority task in the system.
   - The data acquisition task should block waiting for the timer event. Upon receipt of the event it should initiate an analog to digital conversion.

5. Write an interrupt service routine for another of the MC68360 timers. The routine should use the value acquired from the joystick to control the speed and direction of the motor.

   - If the joystick is less than +/-5% from the center of its range the motor should not move.
   - If the joystick is to the right of centre, the motor should be driven clockwise.
   - If the joystick is to the left of centre, the motor should be driven counterclockwise.
   - If the joystick is 5% from center the motor should turn at one revolution every ten seconds.
   - If the joystick is at either end of its range the motor should be driven at 4 revolutions per second.
   - Between these extremes the motor speed should depend linearly on the position of the joystick.

   The motor timer interrupt service routine should update a variable holding the motor position (in steps).

6. Modify the network web server application so that it prints the motor position as well as the converted analog values.

7. Test your program.

7.3 Questions

1. How would you ensure that the motor timer interrupt service routine had the highest priority of all MC68360 interrupts?

2. Why would it be important to give the motor timer interrupt the highest priority?

3. What distinguishes RTEMS directives which can be issued from interrupt handlers from RTEMS directives which can not be issued from interrupt handlers?